import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow.keras.layers import LSTM, Conv1D, Dense, Dropout, Attention, Input, concatenate

from tensorflow.keras.models import Model

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

from scipy.stats import norm

from coot\_optimization import CootOptimizer # Hypothetical library

from gray\_wolf\_optimizer import GWO # Hypothetical library

# PART 1: Data Preprocessing

def preprocess\_data(data, time\_steps=10):

"""

Prepare time-series data for training and testing.

Parameters:

- data: Input time-series data as a numpy array or DataFrame.

- time\_steps: Number of time steps to use for feature extraction.

Returns:

- X, y: Features and target arrays for training/testing.

"""

X, y = [], []

for i in range(len(data) - time\_steps):

X.append(data[i:i + time\_steps])

y.append(data[i + time\_steps])

return np.array(X), np.array(y)

# Normalize data

scaler = MinMaxScaler()

data = scaler.fit\_transform(data.reshape(-1, 1))

X, y = preprocess\_data(data)

# Train-test split

train\_size = int(len(X) \* 0.8)

X\_train, X\_test = X[:train\_size], X[train\_size:]

y\_train, y\_test = y[:train\_size], y[train\_size:]

# PART 2: Building the Hybrid Model

def build\_hybrid\_model(input\_shape):

"""

Build a hybrid time-series prediction model combining CNN, LSTM, and Attention mechanisms.

Parameters:

- input\_shape: Shape of the input data.

Returns:

- model: Compiled hybrid model.

"""

input\_layer = Input(shape=input\_shape)

# CNN for local feature extraction

cnn\_layer = Conv1D(filters=64, kernel\_size=3, activation='relu')(input\_layer)

# LSTM for sequential dependencies

lstm\_layer = LSTM(units=128, return\_sequences=True)(cnn\_layer)

# Attention mechanism

attention\_output = Attention()([lstm\_layer, lstm\_layer])

# Dense output layer

dense\_layer = Dense(units=64, activation='relu')(attention\_output)

output\_layer = Dense(units=1)(dense\_layer)

model = Model(inputs=input\_layer, outputs=output\_layer)

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

return model

# Initialize model

model = build\_hybrid\_model(input\_shape=(X\_train.shape[1], X\_train.shape[2]))

# PART 3: Metaheuristic Hyperparameter Optimization

def optimize\_hyperparameters(model, X\_train, y\_train, optimizer='GWO'):

"""

Optimize hyperparameters using a specified metaheuristic algorithm.

Parameters:

- model: Compiled Keras model.

- X\_train, y\_train: Training data.

- optimizer: Optimization algorithm ('GWO' or 'COA').

Returns:

- best\_model: Model with optimized hyperparameters.

"""

if optimizer == 'GWO':

optimizer\_instance = GWO()

elif optimizer == 'COA':

optimizer\_instance = CootOptimizer()

else:

raise ValueError("Invalid optimizer choice")

# Example hyperparameter bounds

bounds = {

'learning\_rate': (0.0001, 0.01),

'batch\_size': (16, 64),

}

best\_params = optimizer\_instance.optimize(bounds, model, X\_train, y\_train)

model.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=best\_params['learning\_rate']),

loss='mse', metrics=['mae'])

return model

# PART 4: Adaptive Learning with Uncertainty Estimation

class BayesianDropout(tf.keras.layers.Layer):

def \_\_init\_\_(self, rate):

super(BayesianDropout, self).\_\_init\_\_()

self.rate = rate

def call(self, inputs, training=None):

return tf.nn.dropout(inputs, rate=self.rate) if training else inputs

# Replace regular dropout in model with BayesianDropout

# Example: Replace Dropout layers with BayesianDropout in model definition

# PART 5: Training and Evaluation

# Train model

history = model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_split=0.2)

# Evaluate model

predictions = model.predict(X\_test)

rmse = np.sqrt(mean\_squared\_error(y\_test, predictions))

print(f"RMSE: {rmse}")

# PART 6: Interpretation and Visualization

import matplotlib.pyplot as plt

plt.plot(y\_test, label="True Values")

plt.plot(predictions, label="Predictions")

plt.legend()

plt.show()

# Confidence intervals using Bayesian Dropout

# Apply multiple stochastic passes to generate uncertainty estimates

n\_passes = 100

all\_predictions = [model.predict(X\_test, training=True) for \_ in range(n\_passes)]

mean\_prediction = np.mean(all\_predictions, axis=0)

std\_prediction = np.std(all\_predictions, axis=0)

plt.fill\_between(range(len(mean\_prediction)),

mean\_prediction - 1.96 \* std\_prediction,

mean\_prediction + 1.96 \* std\_prediction, color='gray', alpha=0.5)

plt.plot(mean\_prediction, label="Mean Prediction")

plt.plot(y\_test, label="True Values")

plt.legend()

plt.show()